CHAPTER 10

MongoDB Use Cases

"MongoDB: Is it useful for me or not?"

In this chapter, we will provide the much needed connection between the features of MongoDB and the business problems that it is suited to solve. We will be using two use cases for sharing the techniques and patterns used for addressing such problems.

Use Case 1 - Performance Monitoring

In this section, you will explore how to use MongoDB to store and retrieve performance data. You'll focus on the data model that you will be using for storing the data. Retrieving will consist of simply reading from the respective collection. You will also be looking at how you can apply sharding and replication for better performance and data safety.

We assume a monitoring tool that is collecting server-defined parameter data in CSV format. In general, the monitoring tools either store the data as text files in a designated folder on the server or they redirect their output to any reporting database server. In this use case, there's a scheduler that will be reading this shared folder path and importing the data within MongoDB database.

Schema Design

The first step in designing a solution is to decide on the schema. The schema depends on the format of the data that the monitoring tool is capturing.

A line from the log files may resemble Table 10-1.

Table 10-1. *Log File*

Node UUID	IP Address	Node Name	MIB	Time Stamp (ms)	Metric Ve
3beb1a8b-040d-4b46-932a-2d31bd353186	10.161.1.73	corp_xyz_sardar	IFU	1369221223384	0.2

S.G. Edward and N. Sabharwal, *Practical MongoDB*, DOI 10.1007/978-1-4842-0647-8_10

The following is the simplest way to store each line as text:

```
{
_id: ObjectId(...),
line: '10.161.1.73 - corp_xyz_sardar [15/July/2015:13:55:36 -0700] "Interface Util" ...
}
```

Although this captures the data, it makes no sense to the user, so if you want to find out events are from a particular server, you need to use regular expressions, which will lead to full scan of the collection, which is very inefficient.

Instead, you can extract the data from the log file and store it as meaningful fields in MongoDB documents.

Note that when designing the structure, it's very important to use the correct data type. This not only saves space but also has a significant impact on the performance.

For example, if you store the date and time field of the log as a string, it'll not only use more bytes but it'll also be difficult to fire date range queries. Instead of using strings, if you store the date as a UTC timestamp, it will take 8 bytes as opposed to 28 bytes for a string, so it'll be easier to execute date range queries. As you can see, using proper types for the data increases querying flexibility.

You will be using the following document for storing your monitoring data:

```
{
_id: ObjectID(...),
Host:,
Time:ISODate(‘’),
ParameterName:’aa’,
Value:10.23
}
```

■ **Note** The actual log data might have extra fields; if you capture it all, it'll lead to a large document, which is an inefficient use of storage and memory. When designing the schema, you should omit the details that are not required. It's very important to identify which fields you must capture in order to meet your requirements.

In your scenario, the most important information that meets your reporting application requirements is the following:

1. Host
2. Timestamp
3. Parameter
4. Value

Operations

Having designed the document structure, next you will look at the various operations that you need to perform on the system.

Inserting Data

The method used for inserting data depends on your application write concerns.

1. If you are looking for fast insertion speed and can compromise on the data safety, then the following command can be used:

   ```
   >db.perfpoc.insert({Host:"Host1", GeneratedOn: new ISODate("2015-07-15T12:02Z"),
   ParameterName:"CPU",Value:13.13},w=0)
   >
   ```

   Although this command is the fastest option available, since it doesn't wait for any acknowledgment of whether the operation was successful or not, you risk losing data.

2. If you want just an acknowledgment that at least the data is getting saved, you can issue the following command:

   ```
   >db.perfpoc.insert({Host:"Host1", GeneratedOn: new ISODate("2015-07-15T12:07Z"),
   ParameterName:"CPU",Value:13.23},w=1)
   >
   ```

   Although this command acknowledges that the data is saved, it will not provide safety against any data loss because it is not journaled.

3. If your primary focus is to trade off increased insertion speed for data safety guarantees, you can issue the following command:

   ```
   >db.perfpoc.insert({Host:"Host1", GeneratedOn: new ISODate("2015-07-15T12:09Z"),
   ParameterName:"CPU",Value:30.01},j=true,w=2)
   >
   ```

In this code, you are not only ensuring that the data is getting replicated, you are also enabling journaling. In addition to the replication acknowledgment, it also waits for a successful journal commit.

■ **Note** Although this is the safest option, it has a severe impact on the insert performance, so it is the slowest operation.

Bulk Insert

Inserting events in bulk is always beneficial when using stringent write concerns, as in your case, because this enables MongoDB to distribute the incurred performance penalty across a group of insert.

If possible, bulk inserts should be used for inserting the monitoring data because the data will be huge and will be generated in seconds. Grouping them together as a group and inserting will have better impact, because in the same wait time, multiple events will be getting saved. So for this use case, you will be grouping multiple events using a bulk insert.

Querying Performance Data

You have seen how to insert the event data. The value of maintaining data comes when you are able to respond to specific queries by querying the data.

For example, you may want to view all the performance data associated with a specific field, say Host.

You will look at few query patterns for fetching the data and then you will look at how to optimize these operations.

Query1: *Fetching the Performance Data of a Particular Host*

```
> db.perfpoc.find({Host:"Host1"})
{ "_id" : ObjectId("553dc64009cb76075f6711f3"), "Host" : "Host1", "GeneratedOn":
ISODate("2015-07-18T12:02:00Z"), "ParameterName" : "CPU", "Value" : 13.13 }
{ "_id" : ObjectId("553dc6cb4fd5989a8aa91b2d"), "Host" : "Host1", "GeneratedOn":
ISODate("2015-07-18T12:07:00Z"), "ParameterName" : "CPU", "Value" : 13.23 }
{ "_id" : ObjectId("553dc7504fd5989a8aa91b2e"), "Host" : "Host1", "GeneratedOn":
ISODate("2015-07-18T12:09:00Z"), "ParameterName" : "CPU", "Value" : 30.01 }
>
```

This can be used if the requirement is to analyze the performance of a host.

Creating an index on Host will optimize the performance of the above queries:

```
>db.perfpoc.ensureIndex({Host:1})
>
```

Query2: *Fetching Data Within a Date Range from July 10, 2015 to July 20, 2015*

```
>db.perfpoc.find({GeneratedOn:{"$gte": ISODate("2015-07-10"), "$lte": ISODate("2015-07-20")}})
{ "_id" : ObjectId("5302fec509904770f56bd7ab"), "Host" : "Host1", "GeneratedOn"
.............
>
```

This is important if you want to consider and analyze the data collected for a specific date range. In this case, an index on "time" will have a positive impact on the performance.

```
>db.perfpoc.ensureIndex({GeneratedOn:1})
>
```

Query3: *Fetching Data Within a Date Range from July 10, 2015 to July 20, 2015 for a Specific Host*

```
>db.perfpoc.find({GeneratedOn:{"$gte": ISODate("2015-07-10"), "$lte": ISODate("2015-07-20")},
Host: "Host1"})
{ "_id" : ObjectId("5302fec509904770f56bd7ab"), "Host" : "Host1", "GeneratedOn"
.............
>
```

This is useful if you want to look at the performance data of a host for a specific time period.

In such queries where multiple fields are involved, the indexes that are used have a significant impact on the performance. For example, for the above query, creating a compound index will be beneficial.

Also note that the field's order within the compound index has an impact. Let's understand the difference with an example. Let's create a compound index as follows:

```
>db.perfpoc.ensureIndex({"GeneratedOn":1,"Host":1})
>
```

Next, do an explain of this:

```
>db.perfpoc.find({GeneratedOn:{"$gte": ISODate("2015-07-10"), "$lte":
ISODate("2015-07-20")}, Host: "Host1"}).explain("allPlansExecution")
.....................................................................
"allPlansExecution" : [
         {
                 "nReturned" : 4,
                 "executionTimeMillisEstimate" : 0,
                 "totalKeysExamined" : 4,
                 "totalDocsExamined" : 4
                 "indexName" : "GeneratedOn_1_Ho
.....................................................................
                                 "isMultiKey" : false,
                                 "direction" : "forward",
       }]
.....................................................................
```

Drop the compound index, like so:

```
>db.perfpoc.dropIndexes()
{
        "nIndexesWas" : 2,
        "msg" : "non-_id indexes dropped for collection",
        "ok" : 1
}
```

Alternatively, create the compound index with the fields reversed:

```
>db.perfpoc.ensureIndex({"Host":1,"GeneratedOn":1})
>
```

Do an explain:

```
>db.perfpoc.find({GeneratedOn:{"$gte": ISODate("2015-07-10"), "$lte":
ISODate("2015-07-20")}, Host: "Host1"}).explain("allPlansExecution")
{
        ........................................
        "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 4,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 4,
        "totalDocsExamined" : 4,
.................................................
"allPlansExecution" : [ ]
.................................................
}
>
```

You can see the difference in the explain command's output.

Using `explain()`, you can figure out the impact of indexes and accordingly decide on the indexes based on your application usage.

It's also recommended to have a single compound indexes covering maximum queries rather than having multiple single key indexes.

Based on you application usage and the results of the explain statistics, you will use only one compound index on `{'GeneratedOn':1, 'Host': 1}` to cover all the above mentioned queries.

Query4: *Fetching Count of Performance Data by Host and Day*

Listing the data is good, but most often queries on performance data are performed to find out the count, average, sum, or other aggregate operation during analysis. Here you will see how to use the `aggregate` command to select, process, and aggregate the results to fulfil the need of the powerful ad-hoc queries.

In order to explain this further, let's write a query that will count the data per month:

```
>db.perfpoc.aggregate(
... [
... {$project: {month_joined: {$month: "$GeneratedOn"}}},
... {$group: {_id: {month_joined: "$month_joined"}, number: {$sum:1}}},
... {$sort: {"_id.month_joined":1}}
... ]
... )
{ "_id" : { "month_joined" : 7 }, "number" : 4 }
>
```

In order to optimize the performance, you need to ensure that the filter field has an index. You have already created an index that covers the same, so for this scenario you need not create any additional index.

Sharding

The performance monitoring data set is humongous, so sooner or later it will exceed the capacity of a single server. As a result, you should consider using a shard cluster.

In this section, you will look at which shard key suits your use case of performance data properly so that the load is distributed across the cluster and no one server is overloaded.

The shard key controls how data is distributed and the resulting system's capacity for queries and writes. Ideally, the shard key should have the following two characteristics:

- Insertions are balanced across the shard cluster.
- Most queries can be routed to a subset of shards to be satisfied.

Let's see which fields can be used for sharding.

1. **Time field**: In choosing this option, although the data will be distributed evenly among the shards, neither the inserts nor the reads will be balanced.

   As in the case of performance data, the time field is in an upward direction, so all the inserts will end up going to a single shard and the write throughput will end up being same as in a standalone instance.

   Most reads will also end up on the same shard, assuming you are interested in viewing the most recent data frequently.

2. **Hashes**: You can also consider using a random value to cater to the above situations; a hash of the `_id` field can be considered the shard key.

   Although this will cater to the write situation of the above (that is, the writes will be distributed), it will affect querying. In this case, the queries must be broadcasted to all the shards and will not be routable.

3. Use the key, which is evenly distributed, such as **Host**.

   This has following advantages: if the query selects the host field, the reads will be selective and local to a single shard, and the writes will be balanced.

   However, the biggest potential drawback is that all data collected for a single host must go to the same chunk since all the documents in it have the same shard key. This will not be a problem if the data is getting collected across all the hosts, but if the monitoring collects a disproportionate amount of data for one host, you can end up with a large chunk that will be completely unsplittable, causing an unbalanced load on one shard.

4. Combining the best of options 2 and 3, you can have a compound shard key, such as *{host:1, ssk: 1}* where *host* is the host field of the document and *ssk* is _id field's hash value.

   In this case, the data is distributed largely by the host field making queries, accessing the host field local to either one shard or group of shards. At the same time, using ssk ensures that data is distributed evenly across the cluster.

   In most of the cases, such keys not only ensure ideal distribution of writes across the cluster but also ensure that queries access only the number of shards that are specific to them.

Nevertheless, the best way is to analyze the application's actual querying and insertions, and then select an appropriate shard key.

Managing the Data

Since the performance data is humongous and it continues to grow, you can define a data retention policy which states that you will be maintaining the data for a specified period (say 6 months).

So how do you remove the old data? You can use the following patterns:

- **Use a capped collection:** Although capped collections can be used to store the performance data, it is not possible to shard capped collections.
- **Use a TTL collection:** This pattern creates a collection similar to capped collection, but it can be sharded.

  In this case, a time-to-live index is defined on the collection, which enables MongoDB to periodically `remove()` old documents from the collection. However, this does not possess the performance advantage of the capped collection; in addition, the `remove()` may lead to data fragmentation.

1. **Multiple collections to store the data:** The third pattern is to have a day-wise collection created, which contains documents that store that day's performance data. This way you will end up having multiple collections within a database. Although this will complicate the querying (in order to fetch two days' worth of data, you might need to read from two collections), dropping a collection is fast, and the space can be reused effectively without any data fragmentation. In your use case, you are using this pattern for managing the data.

Use Case 2 – Social Networking

In this section, you will explore how to use MongoDB to store and retrieve data of a social networking site.

This use case is basically a friendly social networking site that allows users to share their statuses and photos. The solution provided for this use case assumes the following:

1. A user can choose whether or not to follow another user.
2. A user can decide on the circle of users with whom he wants to share updates. The circle options are Friends, Friends of Friends, and Public.
3. The updates allowed are status updates and photos.
4. A user profile displays interests, gender, age, and relationship status.

Schema Design

The solution you are providing is aimed at minimizing the number of documents that must be loaded in order to display any given page. The application in question has two main pages: a first page that displays the user's wall (which is intended to display posts created by or directed to a particular user), and a social news page where all the notifications and activities of all the people who are following the user or whom the user is following are displayed.

In addition to these two pages, there is a user profile page that displays the user's profile-related details, with information on his friend group (those who are following him or whom he is following). In order to cater to this requirement, the schema for this use case consists of the following collections.

The first collection is `user.profile`, which stores the user's profile-related data:

```
{
_id: "User Defined unique identifier",
UserName: "user name"
ProfilDetaile:
            {Age:.., Place:.., Interests: ...etc},
FollowerDetails: {
                "User_ID":{name:..., circles: [circle1, circle2]}, ....
               },
 CirclesUserList: {
                "Circle1":
                        {"User_Id":{name: "username"}, ......
                        }, .......
               }        ,
      ListBlockedUserIDs: ["user1",...]
}
```

- In this case, you are manually specifying the `_id` field.
- `Follower` lists the users who are following this user.
- `CirclesUserList` consists of the circles this user is following.
- `Blocked` consist of users whom the user has blocked from viewing his/her updates.

The second collection is the user.posts collection, with the following schema:

```
{
_id: ObjectId(...),
by: {id: "user id", name: "user name"},
VisibleTocircles: [],
PostType: "post type",
ts: ISODate(),
Postdetail: {text: "",
Comments_Doc:
 [
{Commentedby: {id: "user_id", name: "user name"}, ts: ISODate(),
Commenttext: "comment text"}, .....
]
}
```

- This collection is basically for displaying all of the user's activities. by provides information on the user who posted the post. Circles controls the visibility of the post to other users. Type is used to identify the content of the post. ts is the datetime when post was created. detail contains the post text and it has comments embedded within it.
- A comment document consists of the following details: by provides details of the user id and name who commented on the post, ts is the time of comment, and text is the actual comment posted by the user.

The third collection is user.wall, which is used for rendering the user's wall page in a fraction of a second. This collection fetches data from the second collection and stores it in a summarized format for fast rendering of the wall page.

The collection has the following format:

```
{
_id: ObjectId(...),
User_id: "user id"
PostMonth: "201507",
PostDetails: [
{
_id: ObjectId(..), ts: ISODate(), by: {_id: .., Name:.. }, circles: [..], type: ....
, detail: {text: "..."}, comments_shown: 3
,comments: [
{by: {_id:., Name:....}, ts: ISODate(), text:""}, ......]
},....]}
```

- As you can see, you are maintaining this document per user per month. The number of comments that will be visible the first time is limited (for this example, it's 3); if more comments need to be displayed for that particular post, the second collection needs to be queried.
- In other words, it's kind of a summarized view for quick loading of the user's wall page.

The forth collection is `social.posts`, which is used for quick rendering of the social news screen. This is the screen where all posts get displayed.

Like the third collection, the fourth collection is also a dependent collection. It includes much of the same information as the `user.wall` information, so this document has been abbreviated for clarity:

```
{
_id: ObjectId(...),
user_id: "user id",
postmonth: '2015_07',
postlists: [ ... ]
}
```

Operations

These schemas are optimized for read performance.

Viewing Posts

Since the `social.posts` and `user.wall` collections are optimized for rendering the news feed or wall posts in a fraction of a second, the query is fairly straightforward.

Both of the collections have similar schema, so the fetch operation can be supported by the same code. Below is the pseudo code for the same. The function takes as parameters the following:

- The collection that needs to be queried.
- The user whose data needs to be viewed.
- The month is an optional parameter; if specified, it should list all the posts of the date less than or equal to the month specified.

```
Function Fetch_Post_Details
(Parameters: CollectionName, View_User_ID, Month)
SET QueryDocument to {"User_id": View_User_ID}
IF Month IS NOT NULL
APPEND Month Filter ["Month":{"$lte":Month}] to QueryDocument
Set O_Cursor = (resultset of the collection after applying the QueryDocument filter)
Set Cur = (sort O_Cursor by "month" in reverse order)
while records are present in Cur
                Print record
End while
End Function
```

The above function retrieves all the posts on the given user's wall or news feed in reverse-chronological order.

When rendering posts, there are certain checks that you need to apply. The following are a few of them.

First, when the user is viewing his or her page, while rendering posts on the wall you need to check whether the same can be displayed on their own wall. A user wall contains the posts that he has posted or the posts of the users they are following. The following function takes two parameters: the user to whom the wall belongs and the post that is being rendered:

```
function Check_VisibleOnOwnWall
(Parameters: user, post)
While Loop_User IN user.Circles List
                If post by = Loop_User
return true
                else
return false
end while
end function
```

The above loop goes through the circles specified in the `user.profile` collection, and if the mentioned post is posted by a user on the list, it returns true.

In addition, you also need to take care of the users on the blocked list of the user:

```
function ReturnBlockedOrNot(user, post)
        if post by user id not in user blocked list
                return true
        else
                return false
endfunction
```

You also need to take care of the permission checks when the user is viewing another user's wall:

```
Function visibleposts(parameter user, post)
if post circles is public
                return true
If post circles is public to all followed users
        Return true
set listofcircles = followers circle whose user_id is the post's by id.

if listofcircles in post's circles
        return true
return false

end function
```

This function first checks whether the post's circle is public. If it's public, the post will be displayed to all users.

If the post's circle is not set to public, it will be displayed to the user if he/she is following the user. If neither is true, it goes to the circle of all the users who are following the logged-in user. If the list of circle is in posts circle list, this implies that the user is in a circle receiving the post, so the post will be visible. If neither condition is met, the post will not be visible to the user.

In order to have better performance, you need an index on `user_id` and `month` in both the `social.posts` and `user.wall` collections.

Creating Comments

To create a comment by a user on a given post containing the given text, you need to execute code similar to the following:

```
Function postcomment(
Parameters: commentedby, commentedonpostid, commenttext)
Set commentedon to current datetime
Set month to month of commentedon
Set comment document as {"by": {id: commentedby[id], "Name": commentedby["name"]},
"ts": commentedon, "text": commenttext}
Update user.posts collection. Push comment document.
Update user.walls collection. Push the comment document.
Increment the comments_shown in user.walls collection by 1.
Update social.posts collection.  Push the comment document.
Increment the comments_shown counter in social.posts collection by 1.
End function
```

Since you are displaying a maximum of three comments in both dependent collections (the `user.wall` and `social.posts` collections), you need to run the following update statement periodically:

```
Function MaintainComments
SET MaximumComments = 3
Loop through social.posts
        If posts.comments_shown > MaximumComments
                Pop the comment which was inserted first
                Decrement comments_shown by 1
        End if
Loop through user.wall
        If posts.comments_shown > MaximumComments
                Pop the comment which was inserted first
                Decrement comments_shown by 1
        End if

End loop
End Function
```

To quickly execute these updates, you need to create indexes on `posts.id` and `posts.comments_shown`.

Creating New Post

The basic sequence of operations in this code is as follows:

1. The post is first saved into the "system of record," the `user.posts` collection.
2. Next, the `user.wall` collection is updated with the post.
3. Finally, the `social.posts` collection of everyone who is circled in the post is updated with the post.

```
Function createnewpost
(parameter createdby, posttype, postdetail, circles)
Set ts = current timestamp.
Set month = month of ts
Set post_document = {"ts": ts, "by":{id:createdby[id], name: createdby[name]},
"circles":circles, "type":posttype, "details":postdetails}
Insert post_document into users.post collection
Append post_document into user.walls collection
Set userlist = all users who's circled in the post based on the posts circle and the posted user id
While users in userlist
Append post_document to users social.posts collection
End while
End function
```

Sharding

Scaling can be achieved by sharding the four collections mentioned above. Since `user.profile`, `user.wall`, and `social.posts` contain user-specific documents, `user_id` is the perfect shard key for these collections. `_id` is the best shard key for the `users.post` collection.

Summary

In this chapter, you used two use cases to look at how MongoDB can be used to solve certain problems. In the next chapter, we will list MongoDB's limitations and the use cases where it's not a good fit.